Chapitre 3

Protocoles de transport de l'Internet

Chapter 3: Couche Transport

Nos objectifs:

- Comprendre les services implémentés par la couche transport:
  - multiplexage/ demultiplexage
  - fiabilité du transfert de données
  - contrôle de flux
  - contrôle de congestion
- Etudier les protocoles de couche transport :
  - UDP: transport sans connexion
  - TCP: transport orienté connexion
  - le contrôle de congestion TCP

Couche transport vs. Couche réseau

- *La couche réseau :* communication logique entre les hôtes

- *La couche transport:* communication logique entre les processus
  - apporter des améliorations de fiabilité par rapport à celle de la couche réseau

Multiplexage/demultiplexae

Demultiplexage au hôte récep:

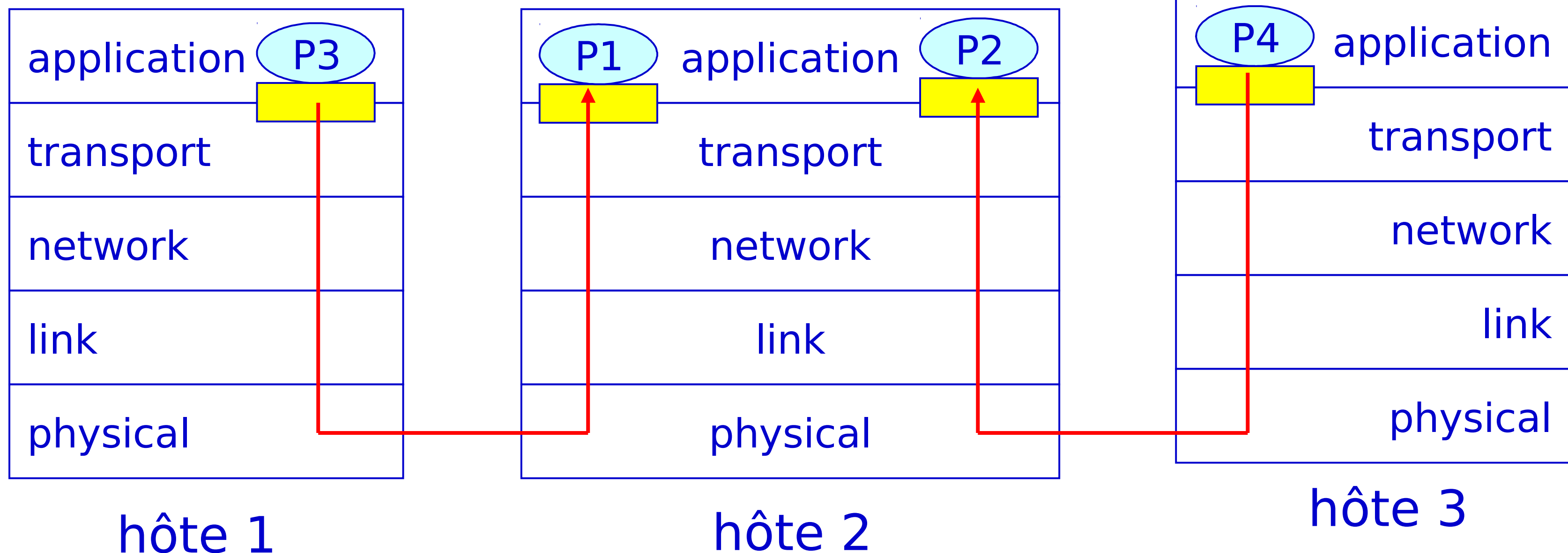
Délivrer les segments à la bonne socket

Multiplexage au hôte émett:

Colleter les données de plusieurs sockets et enveloppe les données avec l'en-tête (utile pour le démultiplexage)

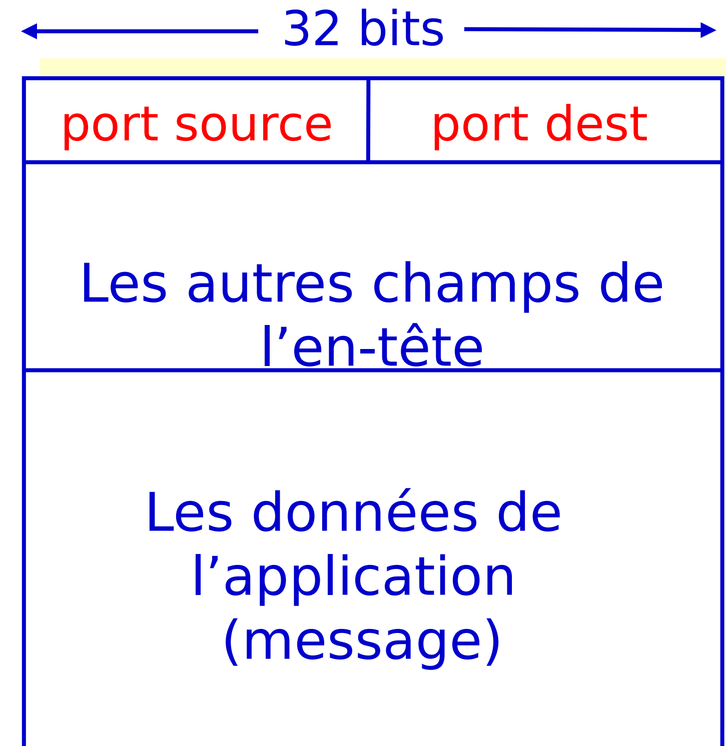
= socket

= process



Comment s'effectue le démultiplexage

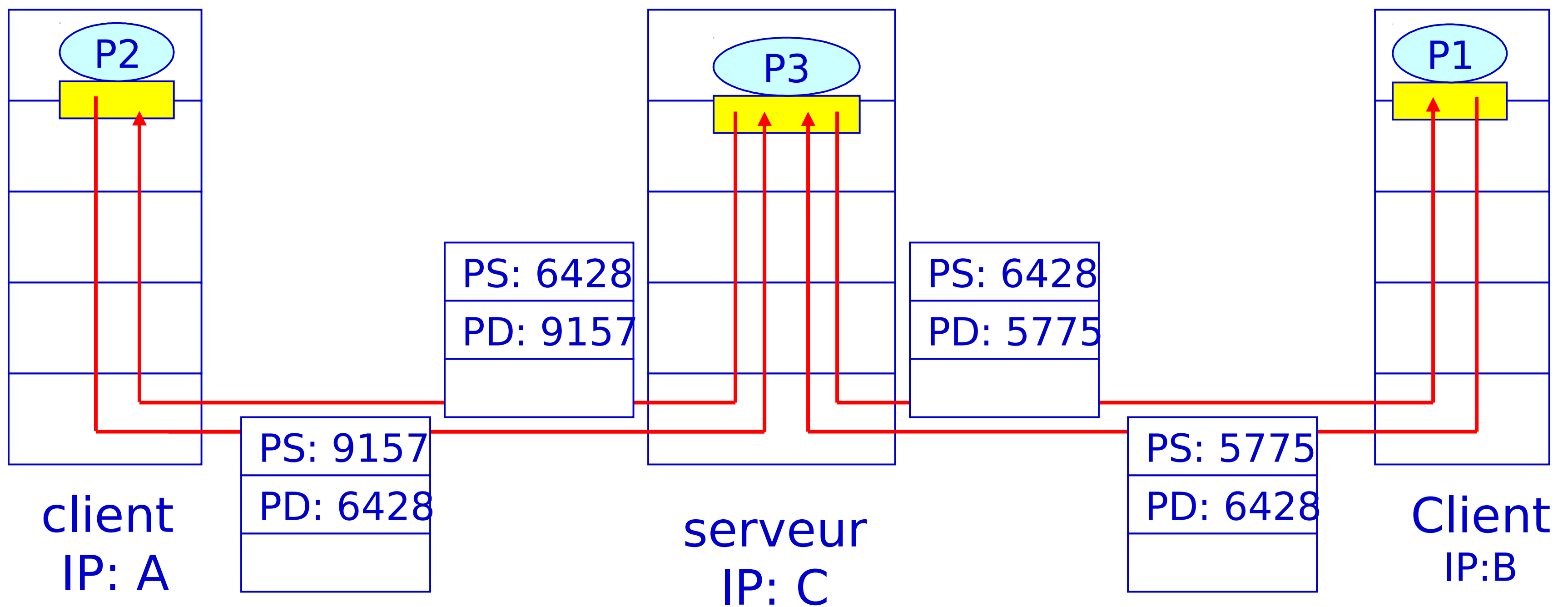
- Hôte recevant les datagramme IP
  - Chaque datagramme a une adresse IP source et une adresse IP destination
  - chaque datagramme encapsule un segment de couche transport
  - Chaque segment a un numéro de port source et un num de port destination spécifiques à l'application
- hôte utilise les adresses IP et les numéros de port pour diriger le segment à la bonne socket



Format d'un segment TCP/UDP

Démultiplexage sans connexion

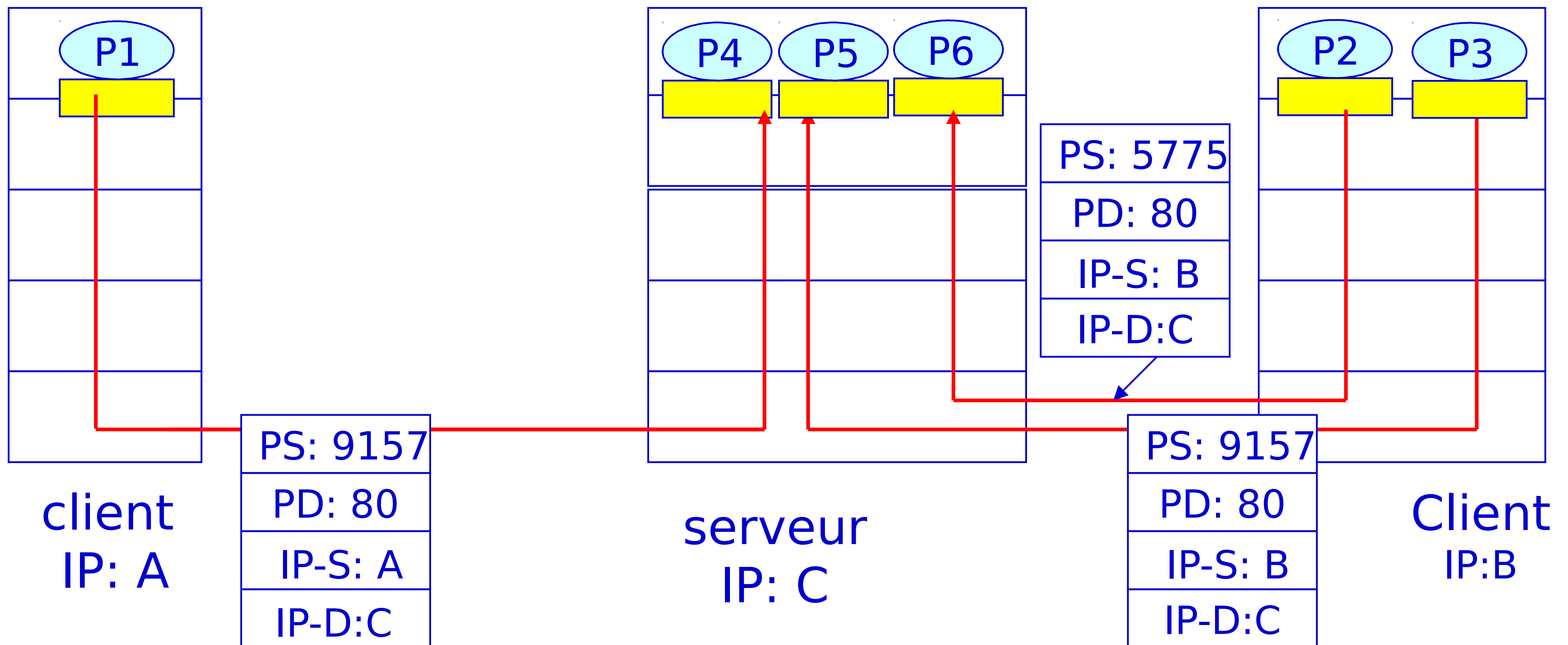
```
DatagramSocket serverSocket = new DatagramSocket(6428);
```



Démultiplexage orienté connexion

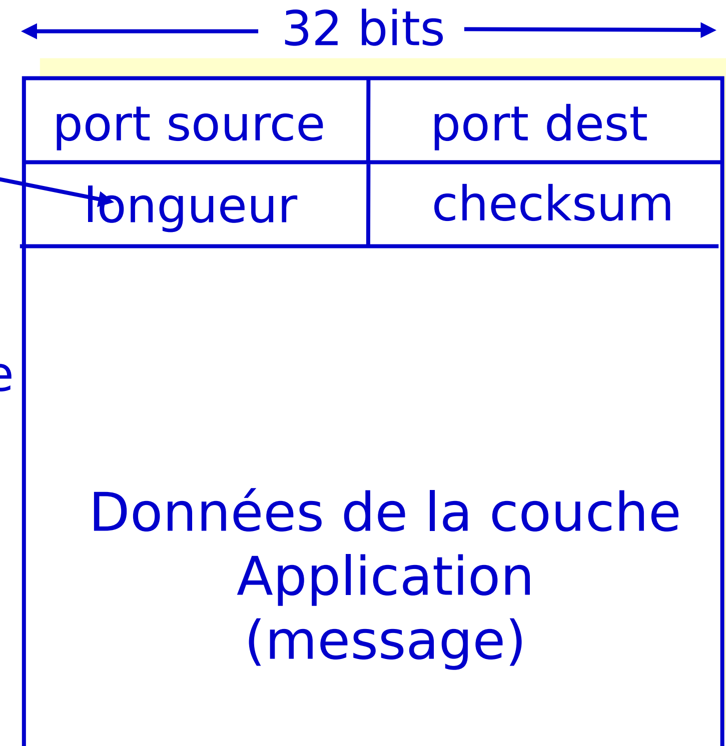
- socket TCP identifiée par 4-tuple:
  - Adresse IP source
  - num de port source
  - Adresse IP dest
  - num de port dest
- l'hôte récepteur utilise toutes les 4 valeurs pour diriger le segment à la socket appropriée
- L'hôte serveur peut supporter simultanément plusieurs sockets :
  - chaque socket est identifiée par ses 4-tuple
- Les serveurs Web ont des sockets différents pour chaque connexion d'un client
  - HTTP non-persistant aura une socket différente pour chaque requête

Démultiplexage orienté connexion



Le protocole UDP

- Souvent utilisé pour des app multimédias
  - tolère la perte
  - mais c'est un service rapide
- autres utilisations de UDP
  - DNS
  - SNMP
- fiabilité de transfert à travers UDP: ajouter des fonctions de contrôle au niveau de la couche application pour améliorer la fiabilité
  - recouvrement spécifique d'erreurs au niveau applicatif



Format d'un segment UDP

checksum UDP

But: detecter des "erreurs" (e.g., bits altérés) dans la transmission du segment

Emetteur:

- traite le contenu du segment comme étant une séquence d'entiers de 16 bits
- checksum: le complément à un de la somme de tous les mots de 16 bits du segment en éliminant tout dépassement de capacité
- met la valeur du checksum dans le champs checksum du segment UDP à envoyer

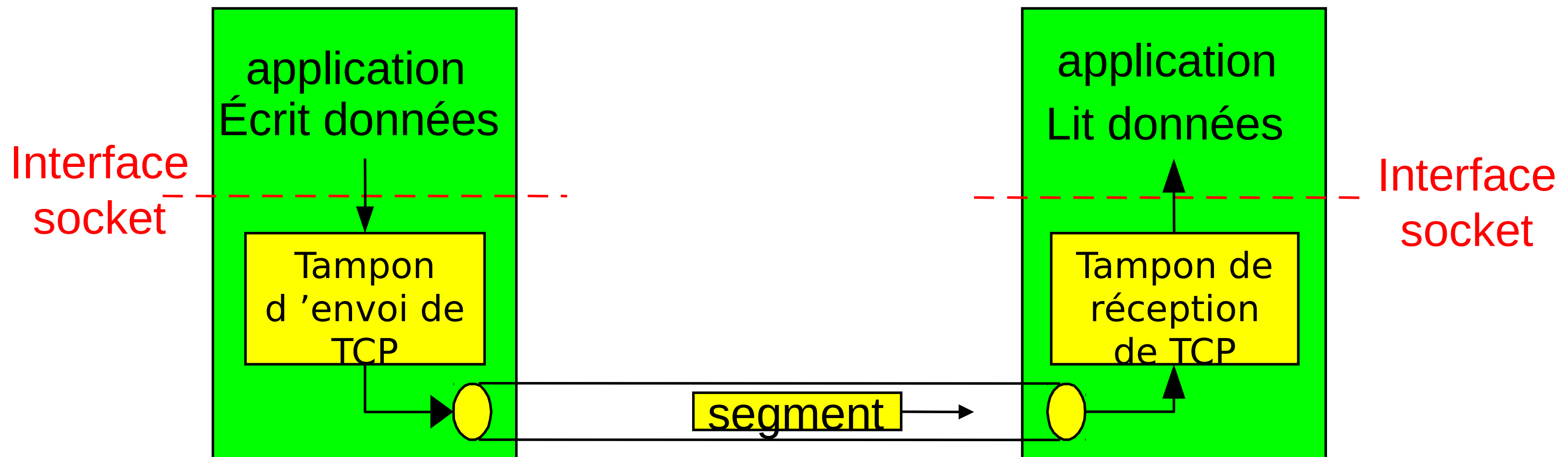
Récepteur:

- Calcule le cheksum du segment reçu
- et vérifie si la valeur calculée du checksum est égale à la valeur mise dans le champ checksum:
  - NON - détection d'erreur
  - OUI - pas d'erreur

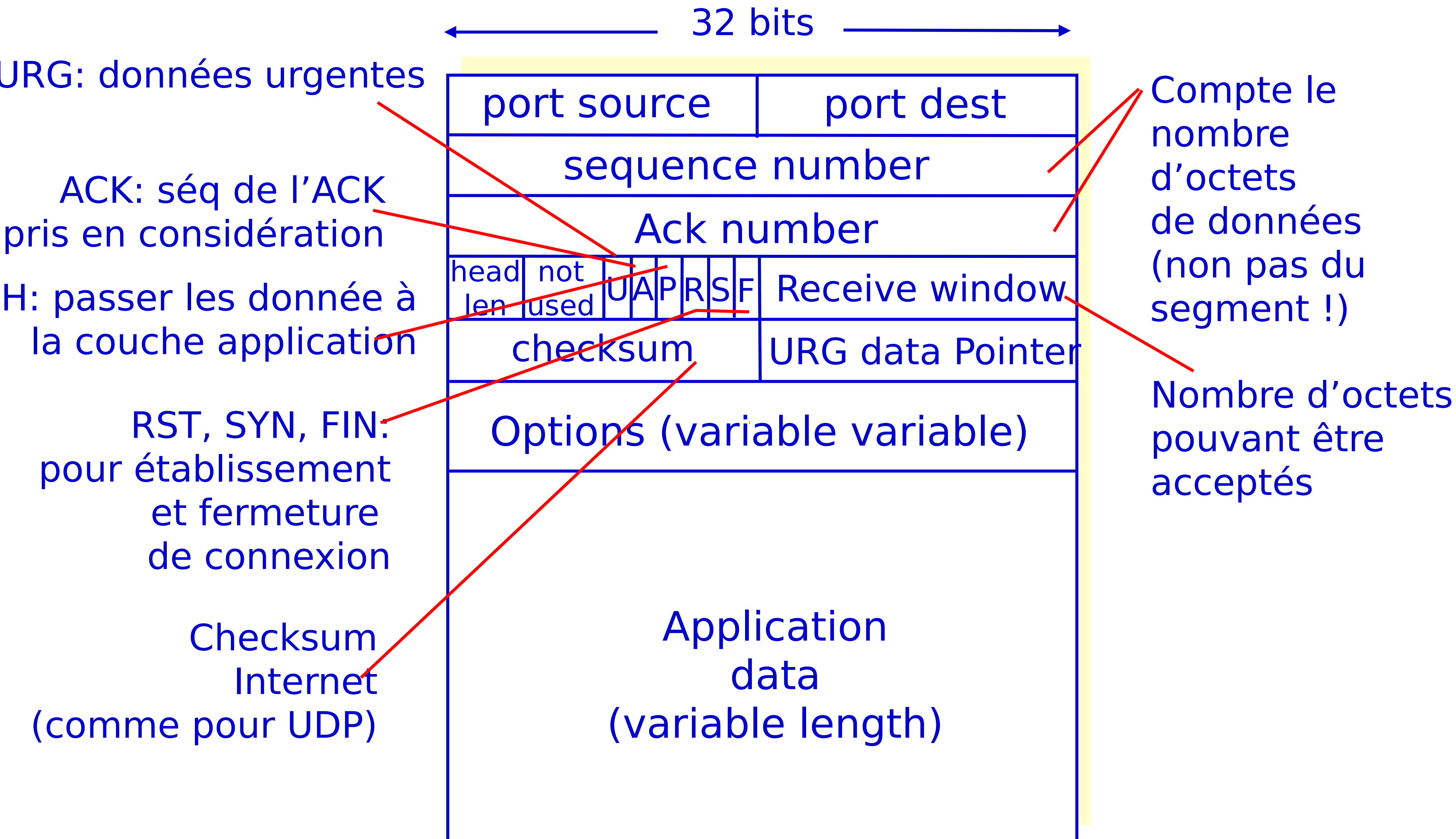
Exemple de checksum Internet

- Note
  - En additionnant les mots de 16 bits, les retenues sortantes du bit du poids fort doivent être ajoutées au résultat final
- Exemple: addition de deux entiers 16-bits

TCP: vue générale
RFCs: 793, 1122, 1323, 2018, 2581



structure d’un segment TCP



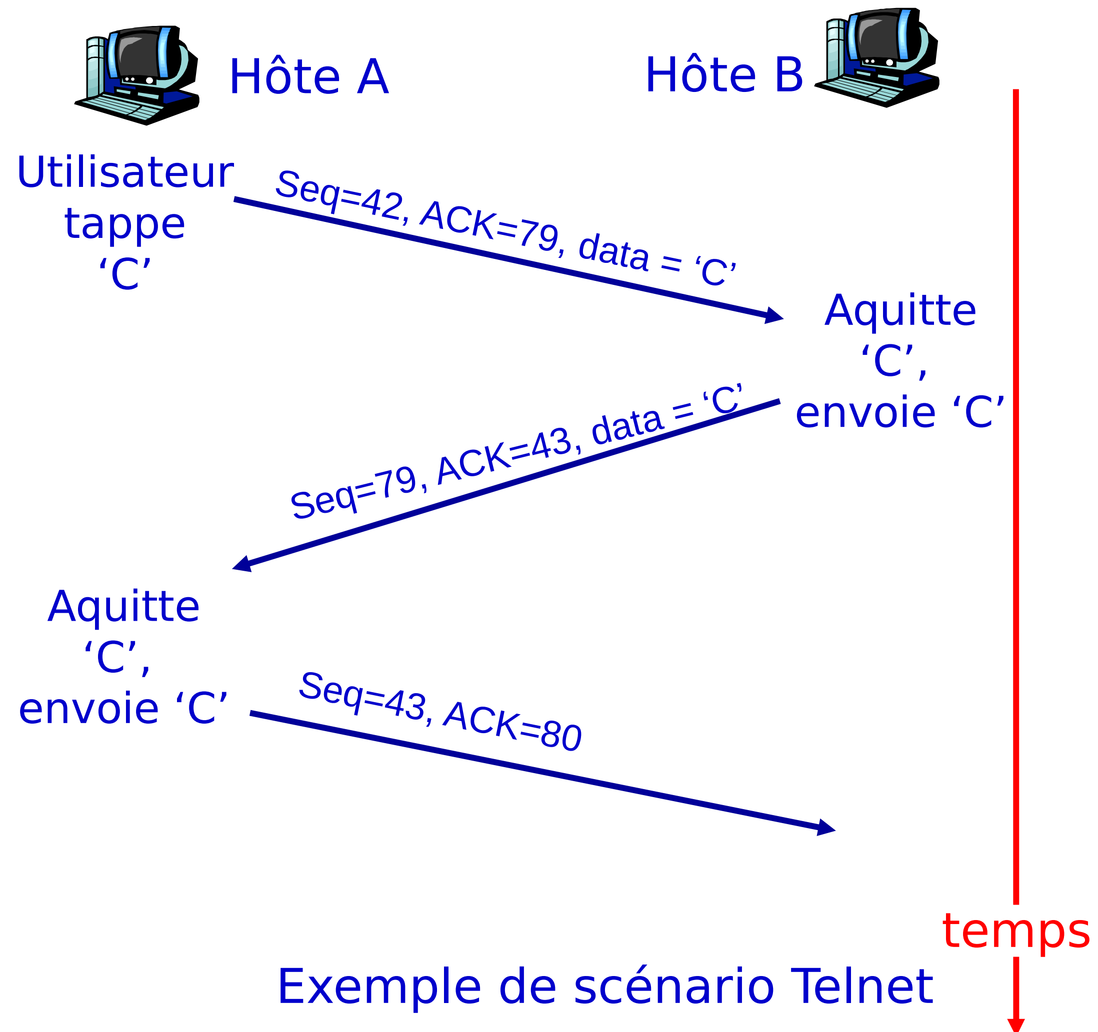
sequence number et acknowledgement number

<u>Numéro de séquence:</u>

- Le numéro dans le flux de données du premier octets de chaque segment

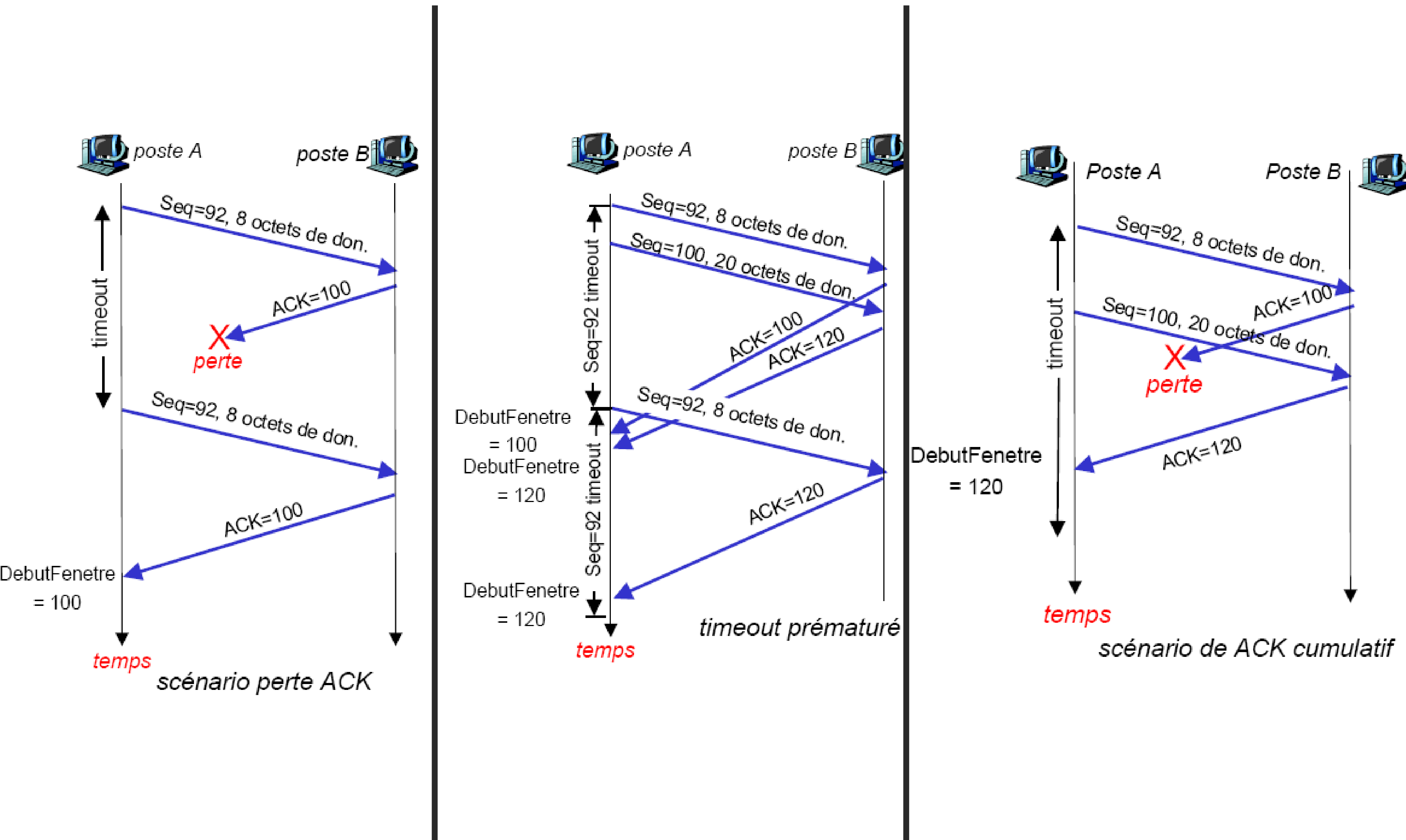
<u>Numéro ACKs:</u>

- le num de seq du prochain octet qu'il attend de l'autre côté
- ACK cumulatif



Exemple de scénario Telnet

Scénarios de retransmission



RTT et Timeout TCP

Q: Comment choisir la valeur de timeout?

- Supérieur à RTT
  - mais RTT varie
- plus petit
  - des retransmissions inutiles
- beaucoup plus grand: ralentit la reaction face à la perte des segments

Q: Comment estimer le RTT?

- **SampleRTT** : mesure le temps entre la transmission du segment et la réception de son ACK
  - ignore les retransmissions
- **SampleRTT** va varier selon le niveau de congestion
  - Ne peut pas être une valeur fiable de RTT
  - On considère une moyenne des mesures et non le **SampleRTT** courant

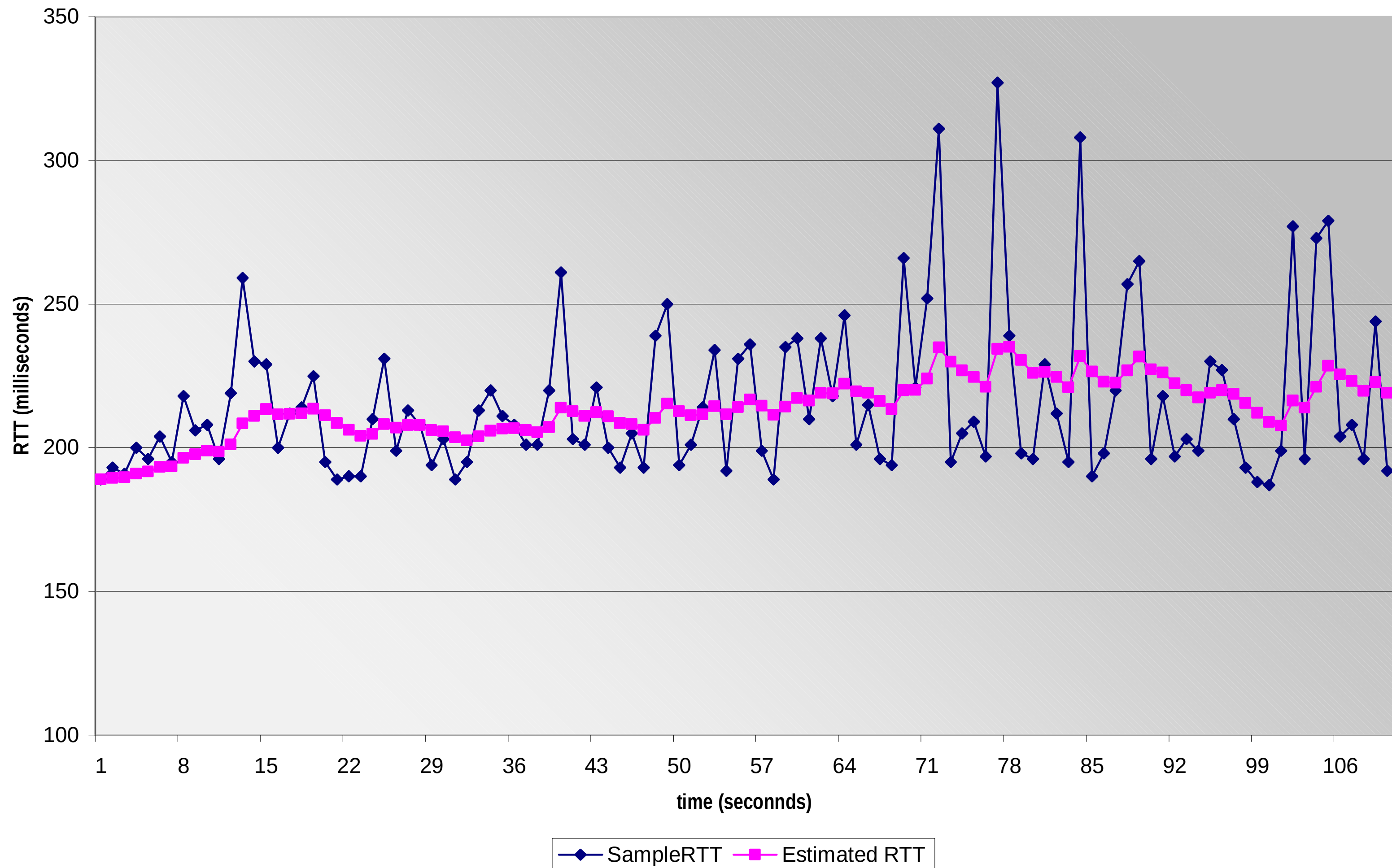
RTT et Timeout TCP

EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

- Une moyenne pondérée des différentes valeurs de **SampleRTT**
- plus de poids aux échantillons récents qu’aux échantillons plus anciens
- valeur typique: α = 0.125

- Timeout= 2 * **EstimatedRTT**

Exemple d’estimation RTT :



RTT et Timeout TCP

Configurtion du timeout (Jacobson & Karels)

- Première estimation de combien SampleRTT dévie de EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
             β*|SampleRTT-EstimatedRTT|

(typiquement, β = 0.25)
```

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

Transfert de données fiable avecTCP

- TCP crée un service de transfert fiable au dessus du service IP non fiable
- Piplinage des segments (plusieurs envoi sans ack)
- acks cumulatifs
- TCP utlise un seul timer de retransmission
- Initialement considérons la version simplifié de TCP
  - ignorer la dupplication des acks
  - ignorer, le contrôle de flux et le contrôle de congestion

Evenements de l'expéditeur TCP:

Données reçues de l'app:

- créer un segment avec un numéro de séquence
  - le num de séq est le numéro dans le flux de données du premier octet de chaque segment
- actionner le timer s'il n'est pas encore déclenché (pensez que le timer est pour le plus ancien segment non acquitté)
- interval d'expiration: `TimeOutInterval`

timeout:

- retransmettre le segment qui a causé le déclenchement du timeout
- redémarrer le timer

réception d'un Ack:

- s'il accuse réception des segments non acquittés
  - mettre à jour ce qu'il n'est pas encore acquitté
  - démarrer le timer s'il y a des segments envoyés non encores acquittés

Vérsion simplifiée de l'expéditeur TCP:

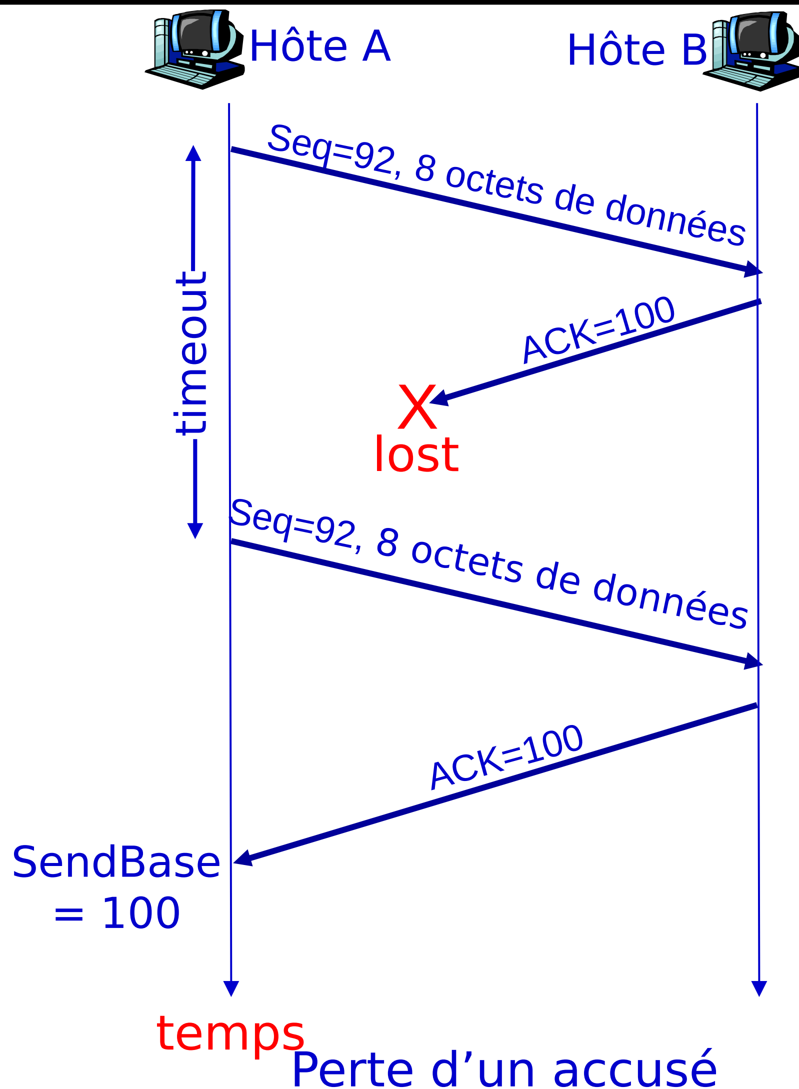
```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
 loop (forever) {
    switch(event)
    event: données reçues de la couche application
 créer un segment TCP avec le num de séq= NextSeqNum

         if (timer n'est pas encore déclenché)
              start timer
         pass segment to IP
         NextSeqNum = NextSeqNum + length(data)
     event: timer timeout
 retransmettre le segment non encore acquitté avec le plus
 petit numéro de séquence
 start timer
     event: ACK reçu, avec séq ACK= y
         if (y > SendBase) {
              SendBase = y
             if (S'il reste des segments non encore acquittés)
                    start timer
             }
}  /* end of loop forever */
```

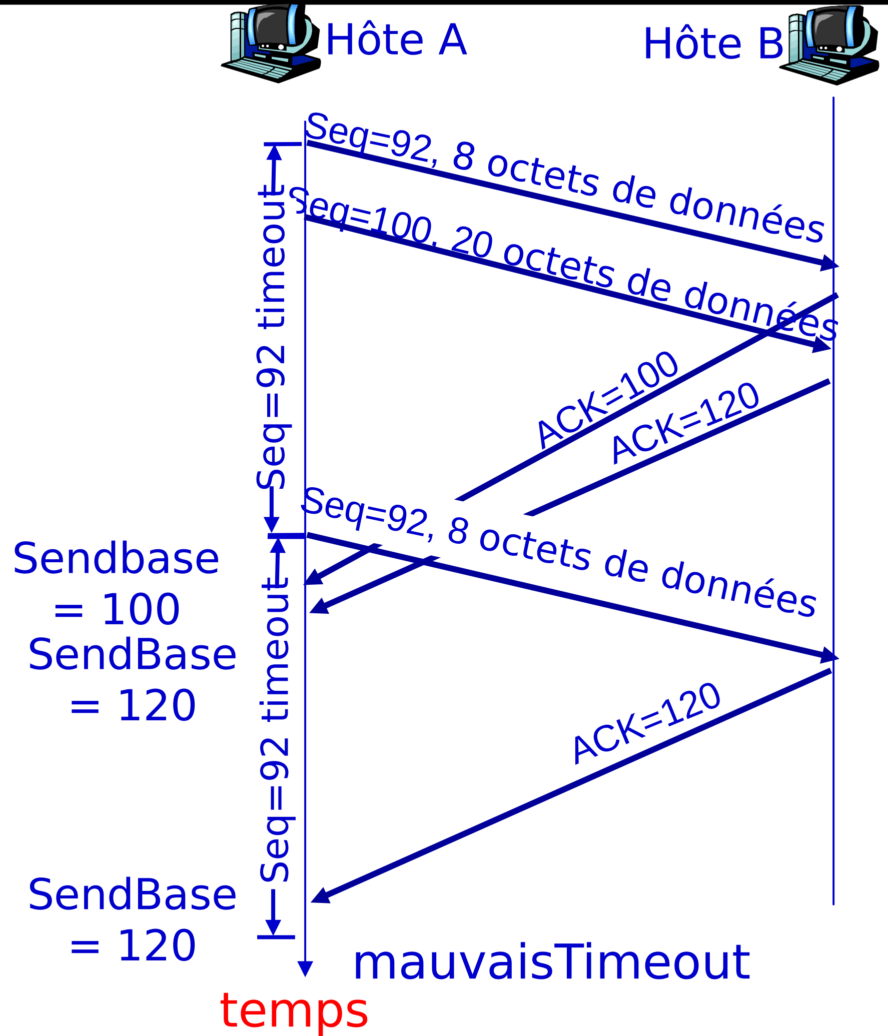
<u>Commentaire:</u>

- SendBase-1: num de séq du dernier octet reçu intact et dans l'ordre
- <u>Example:</u>
- SendBase-1 = 71; y= 73, récepteur demande 73+ ; y > SendBase, don de nouvelle données sont acquittées

TCP: scénarios de retransmission

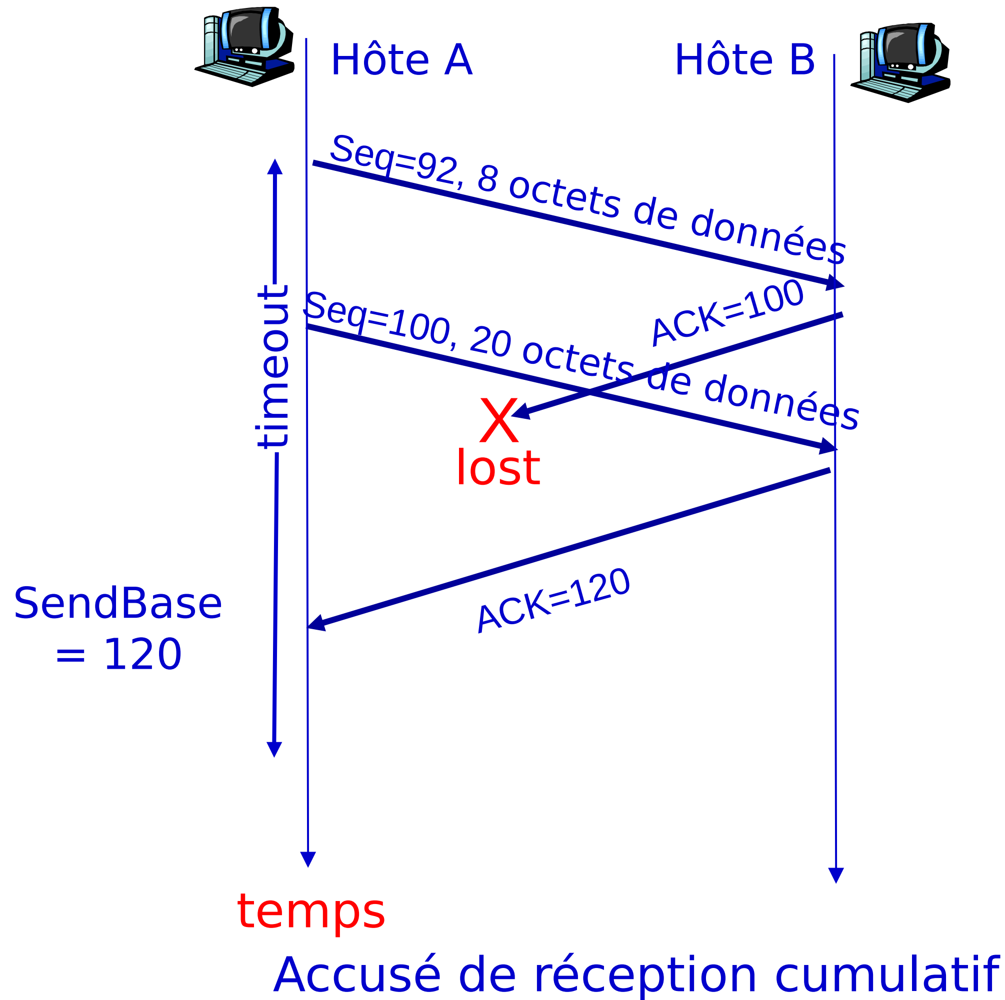


Perte d'un accusé



mauvaisTimeout

TCP: scénarios de retransmission



Accusé de réception cumulatif

Génération des ACK TCP [RFC 1122, RFC 2581]

Evénement	**Réponse du destinataire TCP**
Arrivée d'un segment intact et dans l'ordre, porteur du num de séq attendu. Toutes les données jusqu'à ce num de séq sont déjà acquitté	Génération de l'accusé de réception retardé d'une durée de 500 ms, en attente d'un autre segment
Arrivée d'un segment intact et dans l'ordre, doté d'un num de séq attendu. Un segment précédent est en attente de l'émission de son ACK	Envoi immédiat d'un ACK cumulatif simple, accusant réception des deux segments (dans l'ordre)
Arrivée d'un segment hors séq doté d'un num de Seq supérieur au num attendu. (Lacune détectée)	Envoi immédiat d'un ACK dupliqué indiquant le num de séq du prochain octet attendu (limite inf lacune)
Arrivée d'un segment remplissant complétement ou partiellement la lacune dans les données reçues	Envoi immédiat d'un ACK, sous réserve que le segment coïncide avec la limite inf de la lacune

Retransmission rapide

- Souvent, la periode du Time-out est relativement longue :
  - Délai long avant la retransmission d'un paquet perdu
- Détection de perte de segment avec les ACKs dupliqués.
  - Emétteur envoie souvent un grand nombre de segment l'un derrière l'autre
  - La perte d'un segment est susceptible de générer un grand nombre d'accusé de réception en chaîne.
- Si l'émetteur reçoit 3 ACKs du même segment, il suppose que ce segment est perdu:
  - Retransmission rapide: retransmet le segment avant l'expiration du Time-out

Algorithme de retransmission rapide

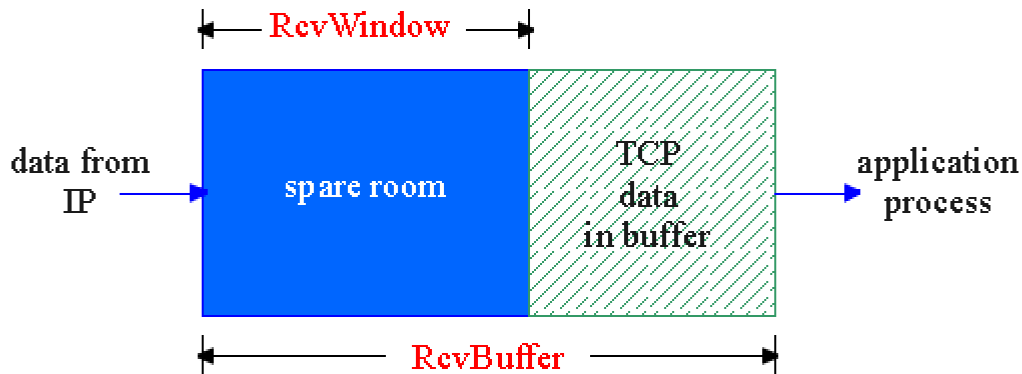
```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
                 SendBase = y
                 if (des segments ne sont pas encore acquittés)
                      start timer
               }
            else {
             incrémenter le compteur des ACKs reçus pour y
                  if (compteur des ACKs reçus pour y = 3) {
           retransmettre le segment avec le num de séq = y
}
```

Un ACK dupppliqué
pour un segment
Déjà acquitté

Retransmission rapide

Contrôle de flux TCP

- Le côte récepteur a un buffer de réception:



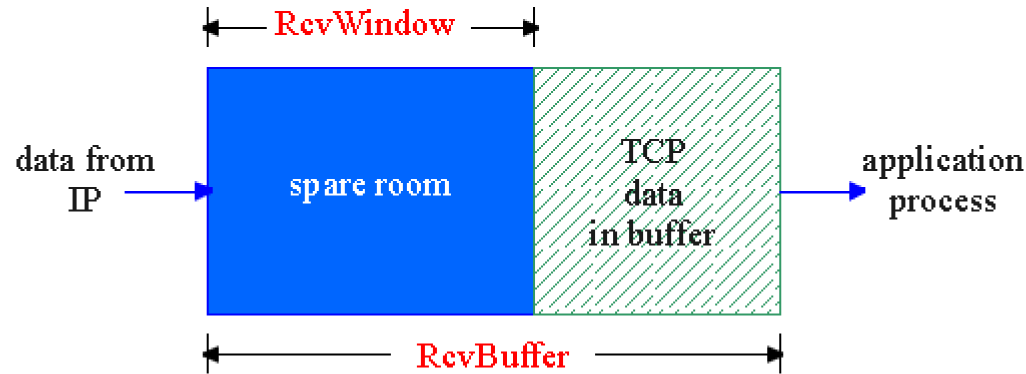
- Le processus applicatif peut prendre du temps (occupée par d'autres activités) pour lire le contenu du buffer

Contrôle de flux

L'emetteur ne doit pas saturer le buffer du récepteur en transmettant avec un rythme trop soutenu

- Eviter la saturation par un système d'équilibrage qui ajuste le rytme d'envoi à la vitesse de lecture de l'application destinataire

Contrôle de flux TCP : comment ça marche

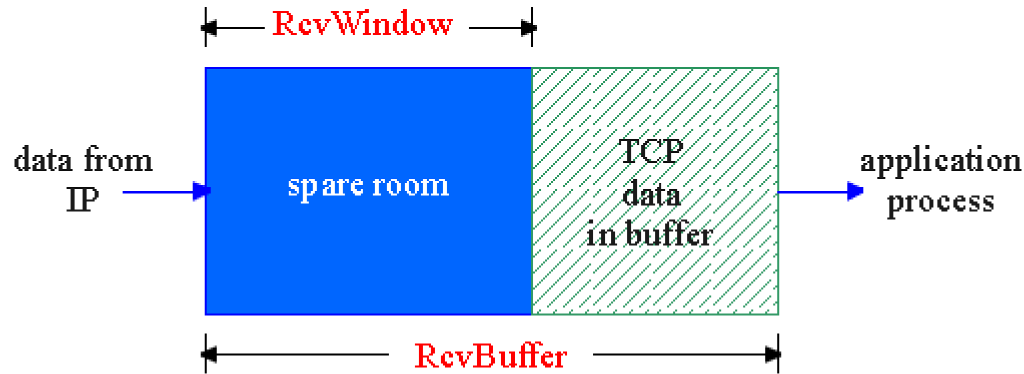


RcvWindow = RcvBuffer-[LastByteRcvd - LastByteRead]

(On suppose que le récepteur TCP supprime les segments hors séquence)

- RcvWindow : Espace mémoire disponible (spare room) en buffer
- RcvBuffer : la taille du buffer de récéption allouée pour la connexion
- LastByteRead : numéro du dernier octet dans le flux de données retiré du buffer par le processus applicatif
- LastByteRcvd : numéro du dernier octet dans le flux de données reçus et placé dans le buffer

Contrôle de flux TCP : comment ça marche



- Récepteur informe l'émetteur de l'espace disponible dans son buffer en insérant **RcvWindow** dans tous les segments TCP.
- L'emetteur limite les segments non acquittés à l'espace **RcvWindow**
- **Si RcvWindow passe à 0, l'émetteur continue à émettre des segments TCP contenant un seul octet de données obligeant le récepteur à envoyer des ACKs porteurs des nouvelles réactualisation de RcvWindow (éviter un deadlock)**

Gestion de la connexion TCP

Initialisation: Echange de segments TCP de connexion entre un client et un serveur

- Initialisation de variables TCP:
  - Num de séq
  - buffers, info contrôle de flux (e.g. **`RcvWindow`**)
- *client:* intitiateur de connexion

```
Socket clientSocket = new
 Socket("hostname","port number");
```

- *serveur:* contacté par le client

```
Socket connectionSocket =
 welcomeSocket.accept();
```

Trois états d'échange:

Etape 1: l'hôte client envoie un segment TCP SYN au serveur

- Spécifie un num de séq initial
- Pas de données

Etape 2: l'hôte serveur reçoit SYN et répond par un segment SYN+ACK

- Serveur alloue les buffers
- Spécifie un num de séq initial serveur

Etape 3: le client reçoit SYN+ACK et répond par un segment ACK qui peut contenir des données

Gestion de la connexion TCP

<u>Fermeture de connexion:</u>

client ferme socket:
`clientSocket.close();`

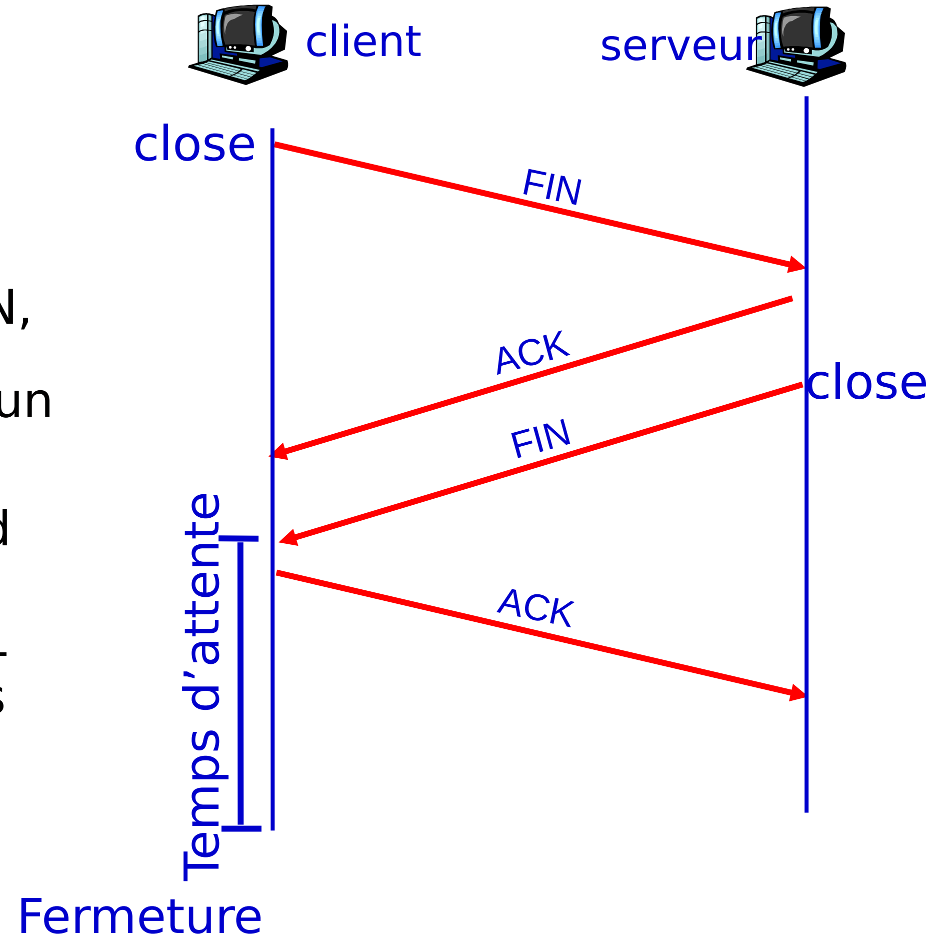
<u>Etape 1:</u> Le système terminal client envoie un segmet de contrôle TCP FIN au serveur

<u>Etape 2:</u> Le serveur reçoit un FIN, répond avec un segment ACK. Ferme la connexion et envoie un segment FIN

<u>Etape 3:</u> client reçoit FIN, répond avec un ACK.

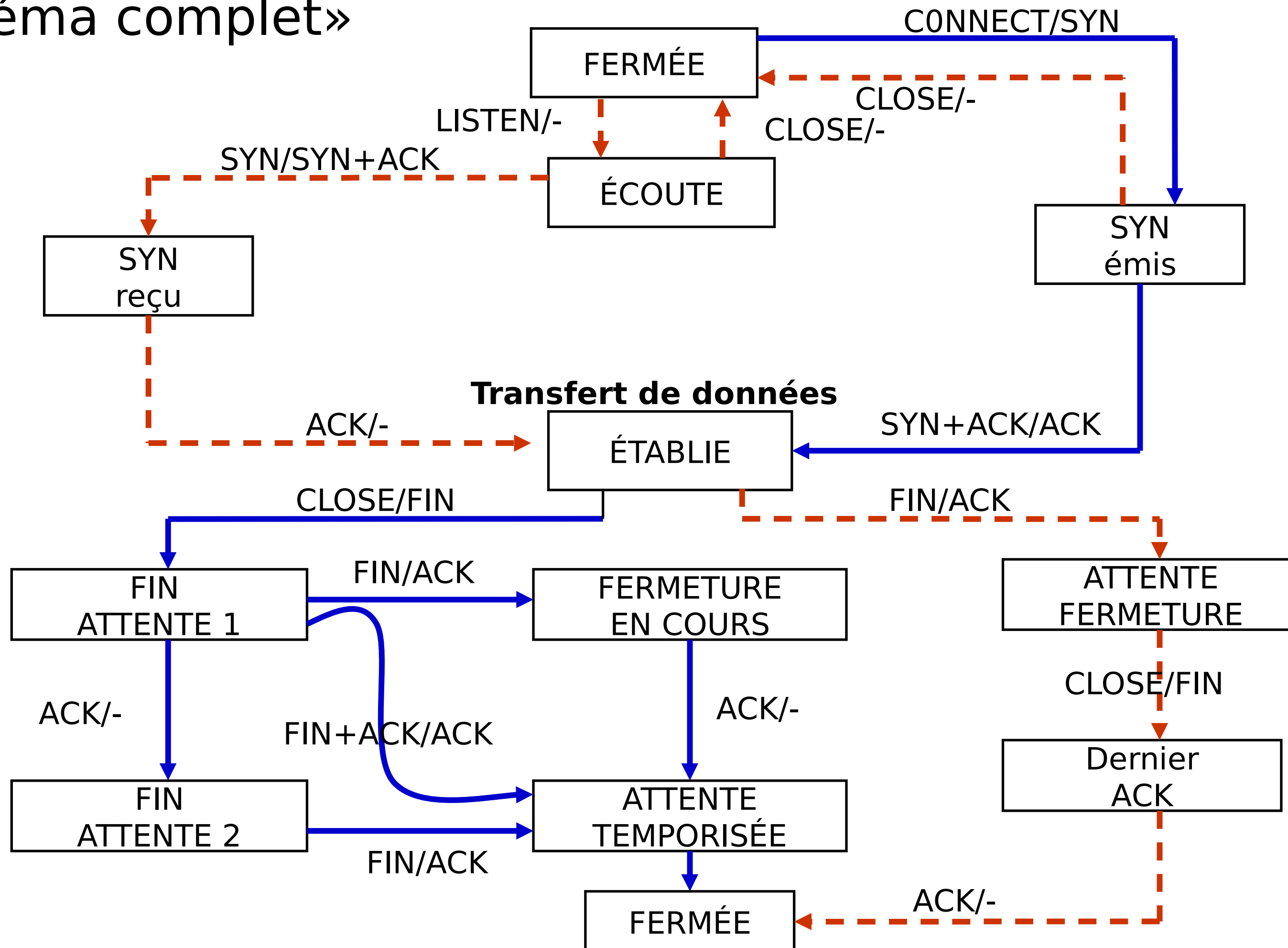
- Met un “temps d’attente” – pour recevoir les segments de données en cours de transfert

<u>Etape 4:</u> serveur, reçoit ACK et ferme la connexion



Gestion de la connexion TCP

- Représentation d'une connexion TCP par un automate à états fini : « schéma complet»



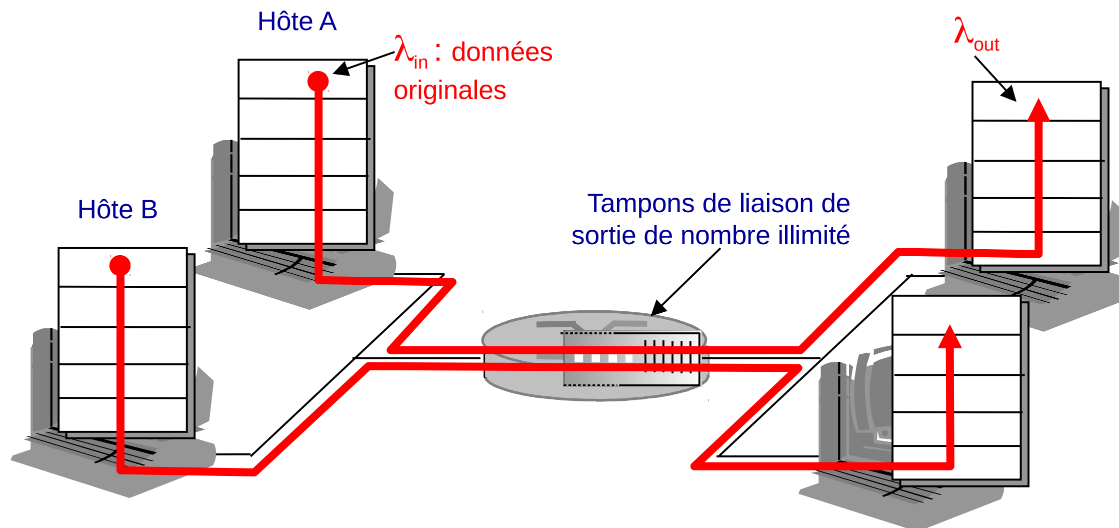
Principe du contrôle de congestion

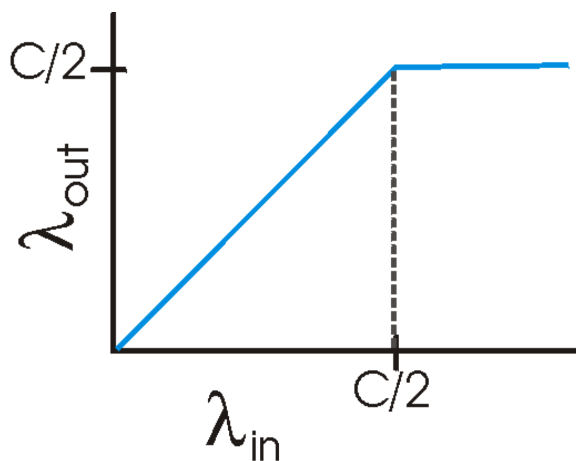
Congestion:

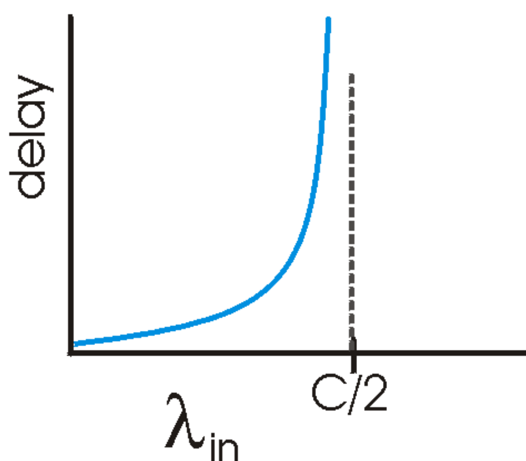
- Informellement :”plusieurs sources envoient un grand volume de données sur un court intervalle de temps au réseau.
- Différent du contrôle de flux !
- Les conséquences:
  - Perte de paquets (saturation des buffers des routeurs “buffer overflow”)
  - Délai de mise en file d’attente très long

Causes et effets de congestion: sénario 1

- Deux émetteurs, deux récepteurs
- Un routeur, buffer de taille infinie
- Pas de retransmission



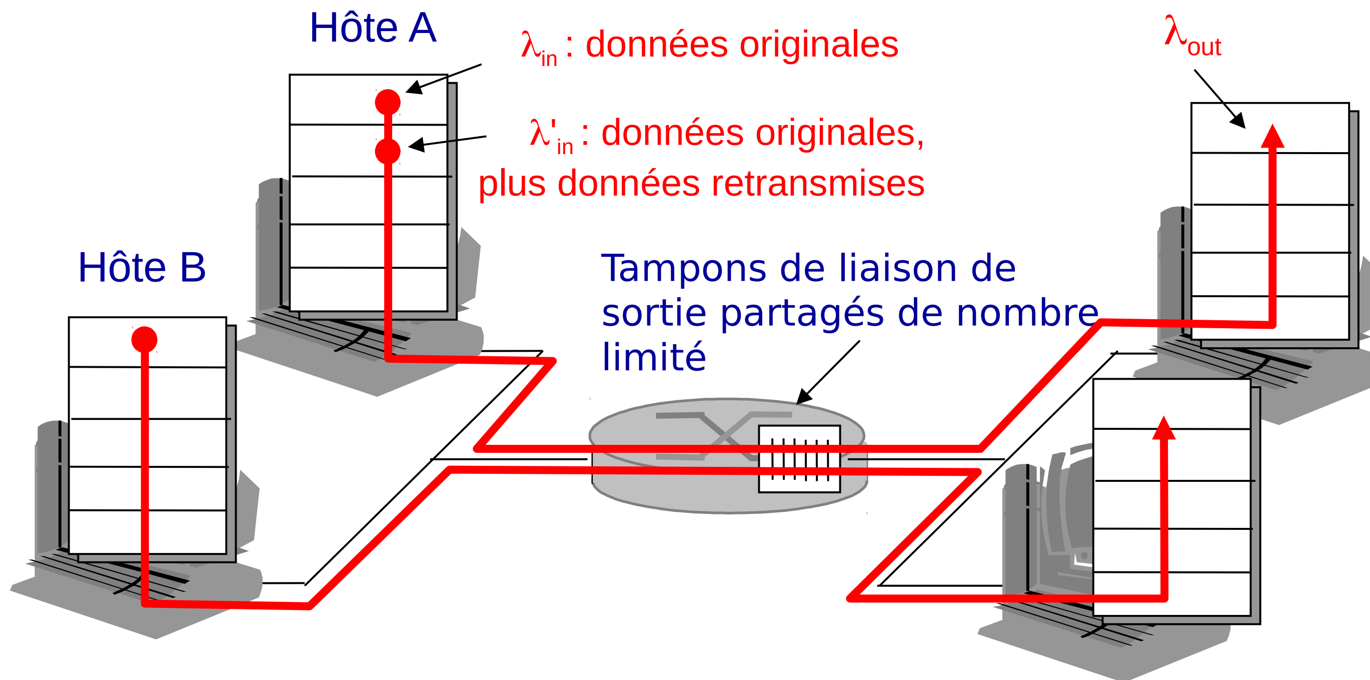




- Un long délai quand c'est congestionné
- Exploitation de la liaison au maximum de sa capacité eq. Longues files d'attentes au niveau du

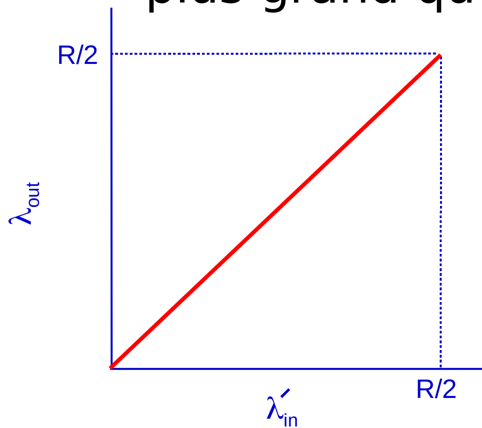
Causes et effets de congestion : scenario 2

- Un seul routeur, buffers limités
- L'émetteur retransmet les paquets perdus

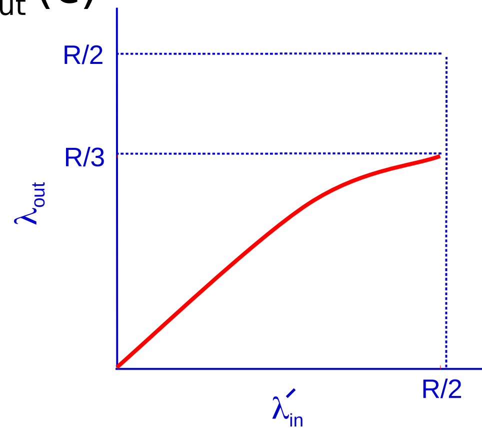


Causes et effets de congestion : scenario 2

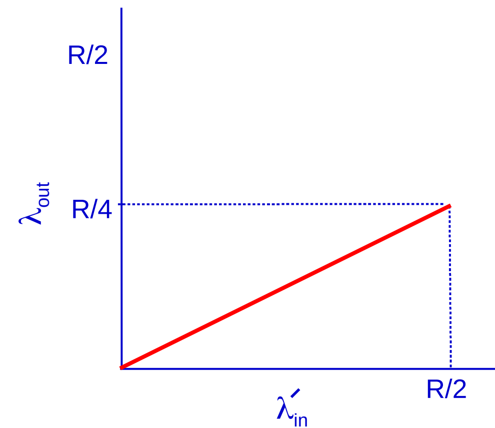
- Dans le cas idéal: $\lambda_{in} = \lambda_{out}$ (a)
- retransmission uniquement dans le cas de perte : $\lambda'_{in} > \lambda_{out}$ (b)
- retransmission des paquets retardé (non perdus) met λ'_{in} encore plus grand que λ_{out} (c)



a.



b.



c.

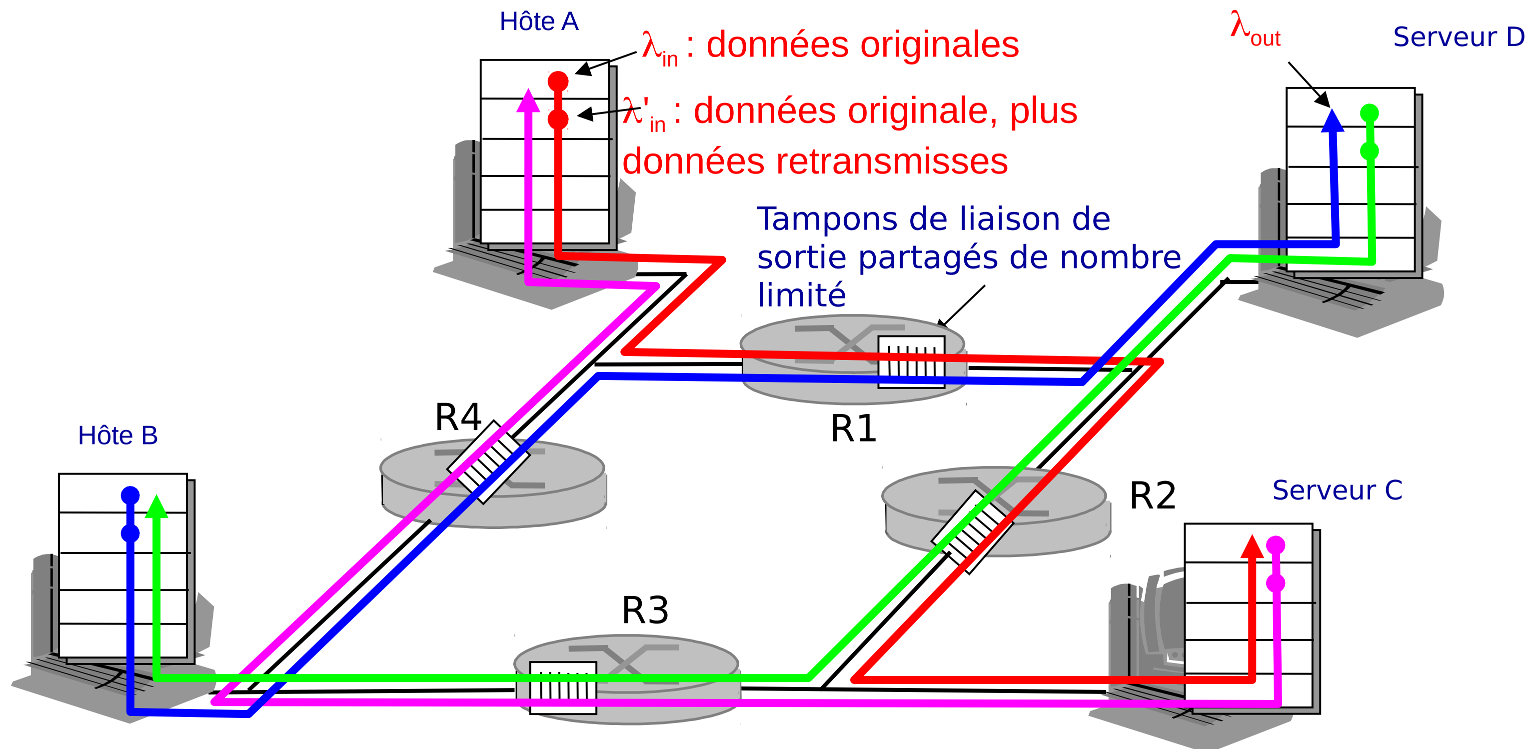
"coût" de congestion:

- Plus de travail (retrasmission) au niveau de l'expéditeur et exploitation inutile du débit de liaison des routeur
- Retramsmission inutile: la liaison supporte de multiples copies des paquets

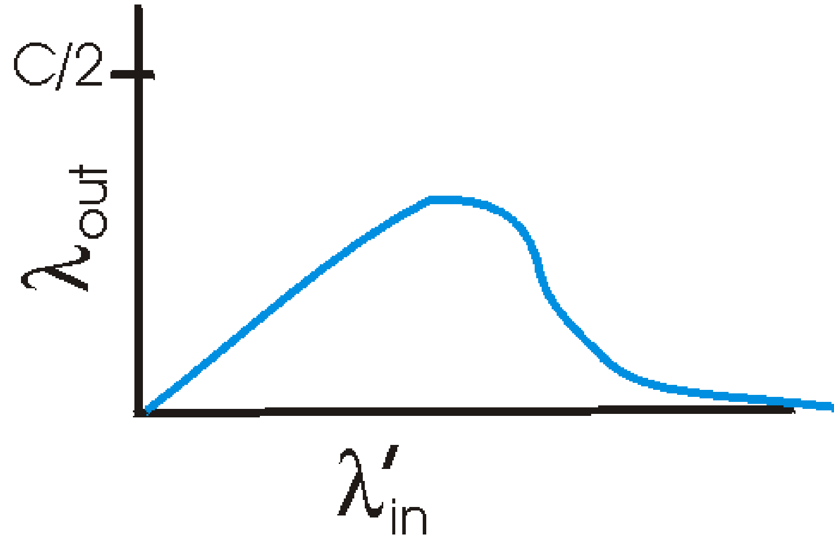
Causes et effets de congestion : scenario 3

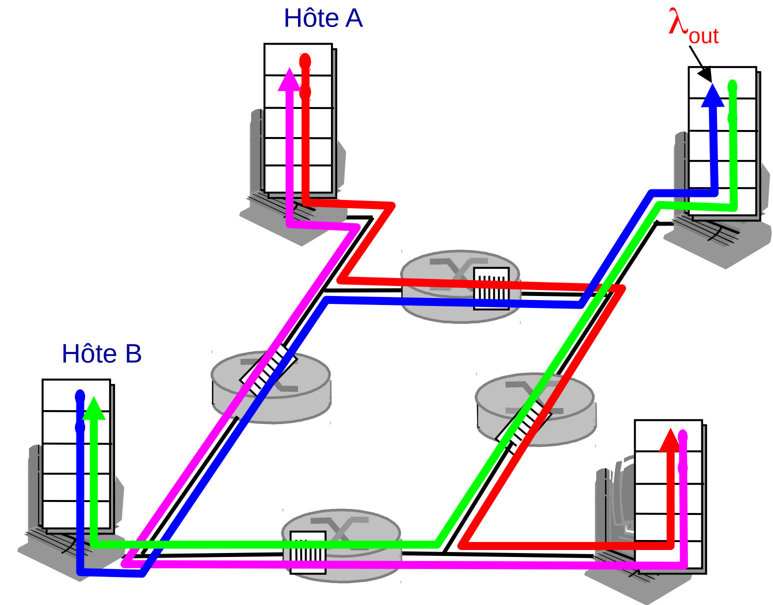
- Quatre émetteurs
- Plusieurs chemins possibles
- timeout/retransmission

Q: Que ce passe –t-il si λ_{in} et λ'_{in} augmentent?



Causes et effets de congestion : scenario 3





Autre “coût” de congestion:

- Lorsqu’un paquet est perdu sur son parcours, la capacité de transmission dépensée par chacun des routeurs en amont pour le faire progresser (jusqu’au routeur saturé) s’en trouve perdue

contrôle de congestionTCP

- Contrôle de bout en bout (pas d'assistance du réseau)
- L'émetteur limite sa transmission:

LastByteSent-LastByteAcked = Min{CongWin,RcvWindows} ≤ CongWin

- Rééllement,

$$\text{Vitesse d'envoi} = \frac{\text{CongWin}}{\text{RTT}} \quad \text{Bytes/sec}$$

- **CongWin** est dynamique et fonction du niveau de congestion du réseau

Comment l'émetteur détecte la congestion?

- Événement de perte = expiration du timeout *ou l'arrivé de* 3 acks duppliqués
- L'émetteur réduit la fenêtre de congestion (**CongWin**) après l'événement de perte

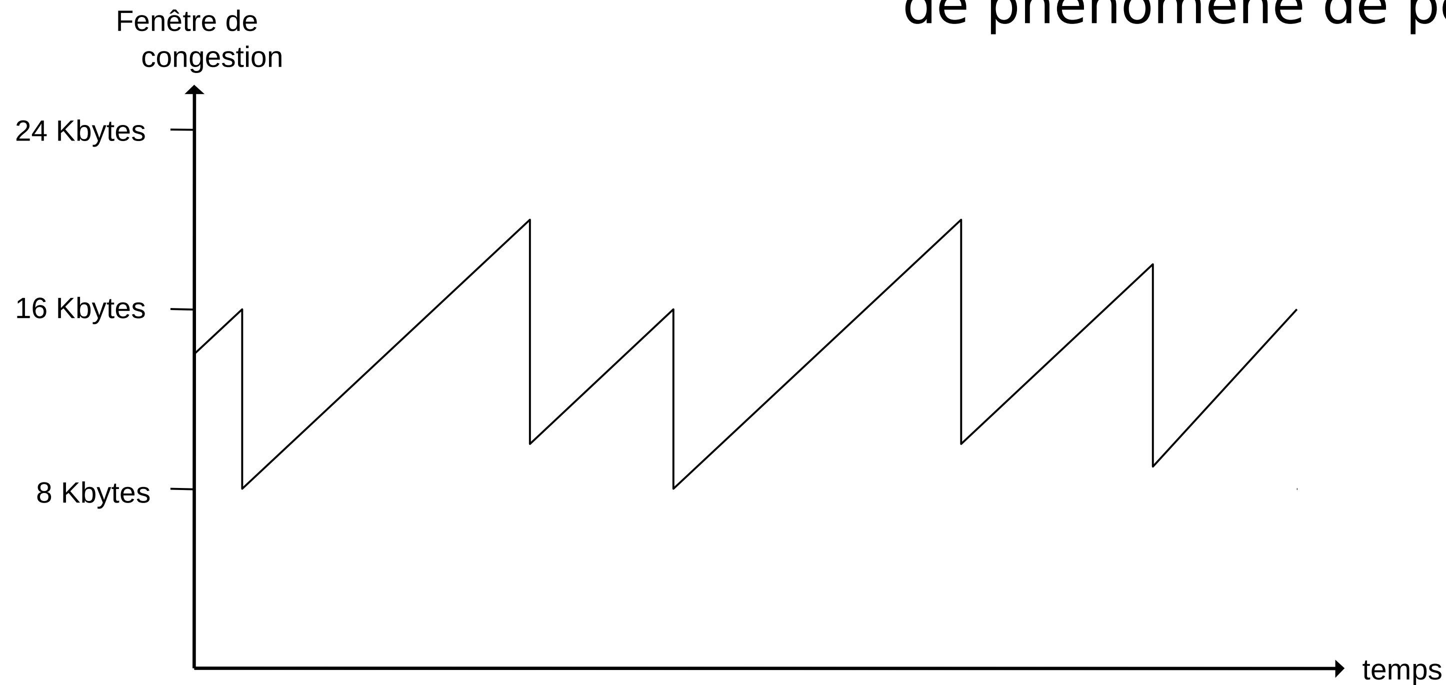
Trois mécanismes:

- Accroissement additif et décroissance multiplicative (AIMD)
- Départ lent
- Réactions aux temporisations

AIMD TCP

décroissance multiplicative: réduire **CongWin** à sa moitié si l'événement de perte apparaît

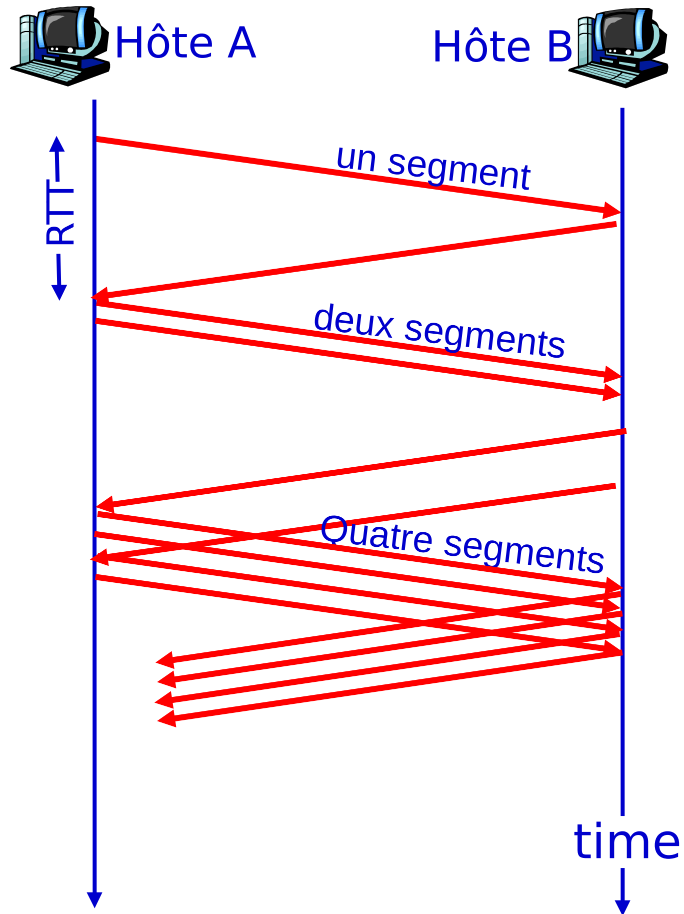
Accroissement additif: augmenter la valeur de CongWin de 1 MSS après chaque RTT en absence de phénomène de perte



Départ lent TCP

- Quand la connexion démarre, **`CongWin`** = 1 MSS
  - Exemple: MSS = 500 bytes & RTT = 200 msec
  - taux d'envoi initiale = 20 kbps
- La bande passante disponible peut être >> MSS/RTT
  - C'est souhaitable de remettre rapidement le taux d'envoi aux taux de la bande passante disponible
- Quand la connexion démarre, le taux d'envoi croît de façon expoentielle jusqu'à l'apparition d'un événement de perte
  - doubler **`CongWin`** chaque RTT
- <u>conclusion:</u> le taux d'envoi initial est lent mais augmente rapidement en exponentiel

Départ lent TCP



Réaction aux temporisations

- Après 3 ACKs duppliqués:
  - **`CongWin`** est divisée à sa moitiée
  - La fenêtre croît linéairement
- Mais après un timeout :
  - **`CongWin`** est mise à 1 MSS;
  - Puis la fenêtre de congestion croît de façon exponentielle jusqu'à un seuil puis elle progresse linéairement
  - La valeur seuil de la fenêtre :
    - Initilialiée à valeur élevée (65 Ko)
    - S'il y a un timeout elle passe à la moitié de CongWin

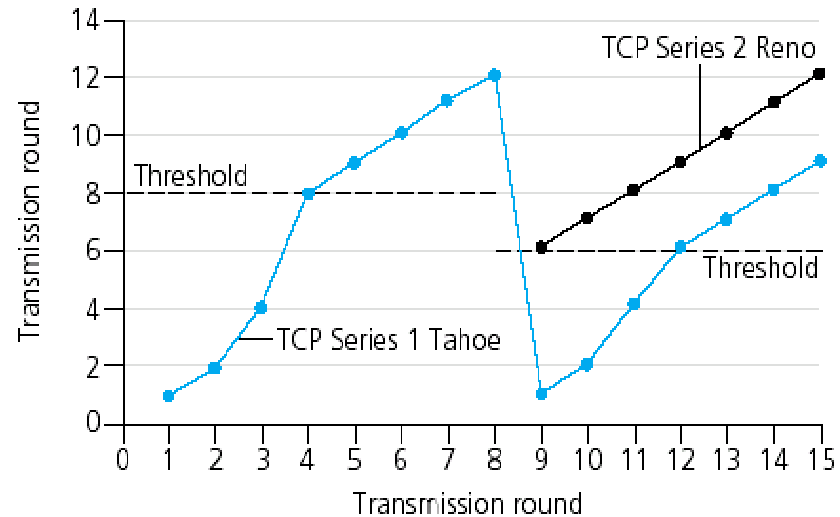
Philosophie:

- 3 ACKs dup indiquent que le réseau est capable de de délivrer quelques segments
- timeout avant 3 ACKs dup est "plus critique"

Réaction aux temporisations

Q: Quand est ce que l'accroissement exponentiel passe au linéaire?

A: Qaund **`CongWin`** prend le 1/2 de sa valeur avant le timeout.



Implementation:

- Variable seuil (Threshold)
- À l'événement de perte, le seuil est mis à 1/2 du CongWin juste avant l'apparition de l'événement de perte

contrôle de congestionTCP : synthèse

- Tant que **CongWin** est inférieure à la valeur du seuil, l’expéditeur est en phase de départ lent et la fenêtre connaît une croissance exponentielle.
- Une fois la valeur de seuil dépassée, l’expéditeur entre dans la phase d’évitement de la congestion durant laquelle la fenêtre s’accroît de manière linéaire.
- Lorsqu’il y a trois ACKs identiques, la valeur du seuil est réglée à la moitié de la taille de **CongWin** en cours et cette dernière est portée à la valeur du seuil
- Lors de l’expiration du timeout, la valeur du seuil est reglée à la moitié de la taille de CongWin en cours et cette dernière est portée à 1 MSS

Contrôle de congestion TCP de l'expéditeur

Evénement	Etat	Action TCP de l'expéditeur	Commentaires
Réception d'un ACK pour un segment non acquitté	Départ Lent (DL)	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to "Evitement de Congestion "	Doubler la valeur de CongWin à chaque RTT
Réception d'un ACK pour un segment non acquitté	Evitement de Congestion (EC)	CongWin = CongWin+MSS * (MSS/CongWin)	Accroissement Additif de CongWin par 1 MSS à chaque RTT
Détection de perte après 3 ACK duplliqués	DL ou EC	Threshold = CongWin/2, CongWin = Threshold, Set state to "Evitement de Congestion "	Recouvrement rapide, par implementation de la décroissance multiplicative. CongWin ne se met pas à 1 MSS.
Timeout	DL ou EC	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Départ Lent "	Passe au Départ Lent
ACK dupppliqué	DL ou EC	Incrémentation du compteur des ACK dup du segment déja acquitté	CongWin et le seuil ne changent pas

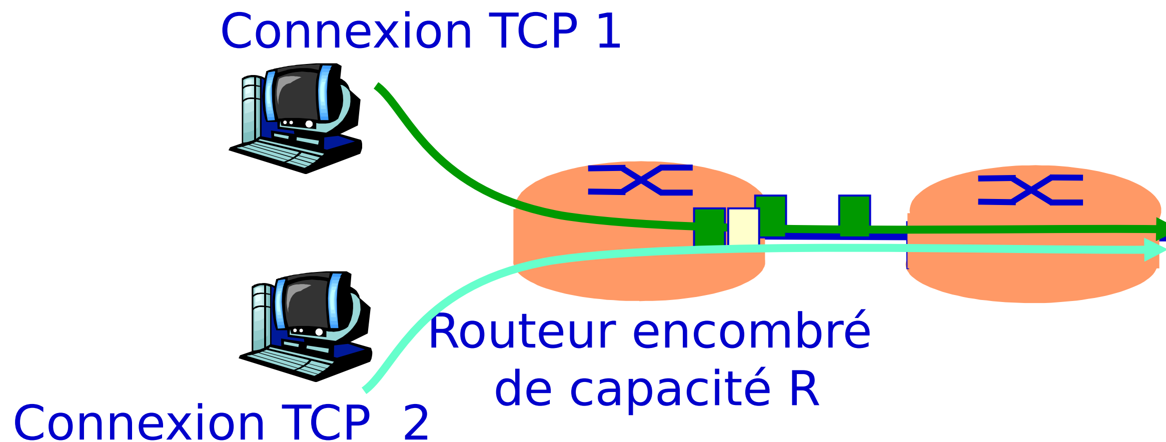
DébitTCP

- Quel est le débit moyen TCP en fonction de la taille de fenêtre et RTT?
  - On Ignore le départ lent
- Soit W la taille de la fenêtre quand il y a perte.
- Quand la fenêtre a la taille W, le débit est W/RTT
- Juste après la perte, la fenêtre passe à W/2 et le débit passe à W/2RTT.

$$\text{Débit moyen} = \frac{0.75 \times W}{RTT}$$

Equité TCP

Objectif de l'équité: Si K sessions TCP partage la même liaison (responsable d'un goulet d'étranglement) de débit R, alors chaqu'une doit avoir un débit moyen de R/K



Comment est implémenté l’equité TCP?

Deux sessions compétitives:

- Acroissement additif ajoute 1 MSS à la fenêtre et le débit croît avec
- Décroissance multiplicatif décroit le débit proportionellement